Today:
• Priority Queues
• Heaps and Heapsort

Data structures
• maintain data
• support various useful operations

Example: Priority Queues
Support these operations:
• INSERT
• EXTRACT-MIN - Remove and return minimum

Applications:
• sorting: insert all elements, extract all elements (in sorted order)
• shortest paths etc (will see later)

Implementations
Priority queues can have many implementations!

E.g., sorted linked-list:
• INSERT - \( \Theta(n) \) time
• EXTRACT-MIN - \( O(1) \)

Heaps
Faster implementation of PQ’s using heaps.

• Binary heap \( A \):
  – Array \( A \) with add’l attribute \( \text{Heap-size}[A] \).
  – Viewed as a nearly complete binary tree
  – heap property:
    \[ A[\text{parent}(x)] \geq A[x] \]
    This defines a \( \text{max-heap} \)
  – There are also min-heaps and \( k \)-ary heaps
Heaps: Example

- Notice the implicit tree links:
  - Children of node $i$ are $2i$ and $2i + 1$
- Why is this useful?
  - Multiplication by 2 is a left shift in binary or add to self – fast
- Which part of tree is empty?

Heaps: Extract-Max

```
HEAP-EXTRACT-MAX(A)
1 \triangleright Removes and returns largest element of A
2   max ← A[1]
3   n ← Heap-size[A]
5   Heap-size[A] ← n − 1
6   HEAPIFY(A, 1) \triangleright Remakes heap
7   return max
```

Running time? $\Theta(1) + \text{HEAPIFY time}$. 

Heaps: Heapify

**HEAPIFY**($A, i$):

- $i$ is index into array $A$
- Binary trees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are heaps
- But, $A[i]$ may be smaller than its children, thus violating the heap property.
- **HEAPIFY** makes $A$ a heap once more.
- How?
  - Move $A[i]$ down in heap until heap property is satisfied.

```
HEAPIFY(A, i)
1 \triangleright Left & Right subtrees of $i$ are heaps.
2 \triangleright Makes subtree rooted at $i$ a heap.
3   l ← LEFT(i) \triangleright $l = 2i$
4   r ← RIGHT(i) \triangleright $r = 2i + 1$
5   if $l \leq n$ and $A[l] > A[i]$
6     then largest ← $l$
7     else largest ← $i$
8   if $r \leq n$ and $A[r] > A[largest]$
9     then largest ← $r$
10    if largest $\neq i$
11      then exchange $A[i] ⇔ A[largest]$
12      HEAPIFY(A, largest)
```
Heaps: Heapify Example

1. Call HEAPIFY(A, 2)


4. Node 9 has no children, so we are done.
Heaps: Building a heap

Remains to show how to implement INSERT. But first, we'll do something simpler.

- Convert an array $A[1..n]$, where $n = \text{length}[A]$, into a heap.
- Notice that the elements in the subarray $A[[\lfloor n/2 \rfloor + 1]..n]$ are already 1-element heaps to begin with.

**BUILD-HEAP($A$)**

1 for $i \leftarrow \lfloor n/2 \rfloor$ downto 1
2 do HEAPIFY($A, i$)

---

**Heaps: BUILD-HEAP: Rough Analysis**

- Correctness: induction on $i$, all trees rooted at $m > i$ are heaps.
- Running time: $n$ calls to $\text{HEAPIFY} = n \cdot O(\lg n) = O(n \lg n)$
- This is good enough for an $O(n \lg n)$ bound on $\text{HEAPSORT}$, but sometimes we build heaps for other reasons, so...

---

**Heaps: BUILD-HEAP: Tighter Analysis**

height of node: longest path from node to leaf

height of tree: height of root

**BUILD-HEAP($A$)**

1 for $i \leftarrow \lfloor n/2 \rfloor$ downto 1
2 do $\text{HEAPIFY}(A, i)$

Time of Heapify = $O($height of subtree rooted at $i$)

Assume $n = 2^k - 1$ (a complete binary tree)

$T(n) = O\left(\frac{n + 1}{2} + \frac{n + 1}{4} \cdot 2 + \frac{n + 1}{8} \cdot 3 + \ldots + 1 \cdot k\right)$

$= O\left((n + 1) \cdot \sum_{i=1}^{k} \frac{i}{2^i}\right)$

$= O(n)$

since

$$\sum_{i=1}^{k} \frac{i}{2^i} = \frac{\frac{1}{2}}{(1 - \frac{1}{2})^2} = 2$$
Heaps: 

**Build-Heap: Tighter Analysis**

How? Use following “trick”:

\[ \sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \text{ if } |x| < 1 \]

Differentiate:

\[ \sum_{i=1}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2} \]

Mult by \( x \):

\[ \sum_{i=1}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2} \]

Plug in \( x = \frac{1}{2} \):

\[ \sum_{i=1}^{\infty} i = \frac{7}{4} = 2 \]

Therefore **Build-Heap** time is \( O(n) \).

---

**Priority Queue: Insertion**

Time for implementation of **Insert**.

Main idea: “heapify upwards”.

**Heap-Insert** \((A, key)\)

1. \( heap\-size[A] \leftarrow heap\-size[A] + 1 \)
2. \( i \leftarrow heap\-size[A] \)
3. **while** \( i > 1 \) and \( A[\text{Parent}(i)] < key \)
4. **do** \( A[i] \leftarrow A[\text{Parent}(i)] \)
5. \( i \leftarrow \text{Parent}(i) \)
6. \( A[i] \leftarrow key \)

Analysis: \( O(\log n) \) time for \( n \) element heap.

---

**Heaps: Heapsort**

\[ \text{Heapsort}(A) \]

1. **Build-Heap** \((A)\)
2. **for** \( i \leftarrow n \) **down to** 2
4. \( \text{Heap-size}[A] \leftarrow \text{Heap-size}[A] - 1 \)
5. \( \text{Heapify}(A, 1) \)

- total running time: \( O(n \log n) \)
- sorts in place!

\( \Rightarrow \) best sorting algorithm so far (at least in theory).